**Java @ Stacks**

**Due date : July 14, 2013 9.00AM (EST)**

**Objectives:**  
  
To gain experience with stacks.  
  
**Documentation:**

1. Explain the purpose of the program as detail as possible -**13%.**
2. Develop a solution for the problem and mention algorithms to be used -**16%**
3. List data structures to be used in solution. - **5%.**
4. Give a description of how to use the program and expected input/output - **5%**
5. Explain the purpose of each class you develop in the program. - **5%.**

**Programming:**

1. For each method, give the pre and post conditions and invariant, if any - **10%**
2. Program execution **according to the requirements given 36%**
3. Naming of program as required **5%**
4. Print out of source code **5%**

**Description of Program**  
  
You are to write a program name **calc.java** that evaluates an infix expression entered by the user. The expression may contain the following tokens:  
(1)   Integer constants (a series of decimal digits).  
(2)   x (representing a value to be supplied later).  
(4)   Binary operators (+, -, \*, / and %).  
(5)   Parentheses  
            
  
Spaces between tokens are allowed but not required. The program will convert the (user input) infix expression to postfix (RPN) form and display the converted expression on the screen.   
The following **example** illustrates the behavior of the program (user input is in **bold and red**):  
Porgram output is in **bold and green.**  
  
Enter infix expression: **(7 + x) \* (8 – 2) / 4 + (x + 2)**  
Converted expression: **7 x + 8 2 - \* 4 / x 2 + +**  
Enter a value for x:**5**  
The result is: **25**  
  
Enter a value for x:**8**  
The result is: **36**  
  
Enter a value for x:**3**  
The result is: **20**  
  
Enter a value for x:**q**------>Any value that is not an integer - the program terminates  
  
When solving the postfix expression, you must **prompt** the user for an input value for **x**, solve the expression and return a result. You must use an operand stack to process the postfix expression.

**Display this result on the screen as well as the postfix expression as shown above.**

If the infix expression contains an error of any kind, the program must display the message Error in expression (with an optional explanation) and then terminate. The following examples illustrate various types of errors:  
  
Enter infix expression: **1 2 +**  
Error in expression!! No operator between operands. Also last token must be an operand.  
  
Enter infix expression: **10.4**  
Error in expression!! Cannot accept floating point numbers.  
  
Enter infix expression: **1 (  + 2)**  
Error in expression!! No operator between operand and left parentheses.  
  
Enter infix expression: **5 – (x – 2))**  
Error in expression!! No matching left parentheses for a right parentheses.  
  
Enter infix expression: **1 \*\* 2**   
Error in expression!! The \* operator cannot be preceded by a \* operator.  
  
The output of your program must match the format illustrated in this example.  
  
Here are some other additional requirements for this program:  
  
(1)    You must use stack objects during the translation from infix to postfix.  
  
(2)    Operators must have the correct precedence and associativity. Binary operators \* , / and % takes precedence over binary + and -. And processing is always done from left to right.

**What to turn in**

Turn in a disk containing **calc.java, calc.class OR Main.java and calc.jar** .

**Hints:**

1.    Do the program in stages. First, get the infix to postfix conversion working for binary operators. Use the examples from the hand out as a starting point for the program, but keep in mind that this code doesn’t handle associativity properly.     
   
3.    To detect errors in the infix expression, you will need to check for several situations:  
  
      A binary operator is preceded by an operator or an operand is preceded by an   
      operand.    
            An illegal character is encountered (such as a period).  
            The last token in the expression is not an operand.  
            There is no left parentheses anywhere in the stack when a right parentheses is   
             encountered.  
            The stack contains a left parenthesis when the expression ends.  
  
4.  Use a string to store the postfix expression. Use a stack of characters   
     during the translation from infix to postfix.  
  
5.  Use the **isdigit()** function to test whether a character is a digit.